Life Cycle Service Specification 6

6.1 Service Description

6.1.1 Overview

Life Cycle Service defines services and conventions for creating, deleting, copying and moving objects. Because CORBA-based environments support distributed objects, the Life Cycle Service defines conventions that allow clients to perform life cycle operations on objects in different locations.

This overview describes the life cycle problem for distributed object systems.

The problem of creation

Figure 6-1 illustrates the problem of a client in one location creating an object in another.

Figure 6-1 Life Cycle service defines how a client can create an object “over there”.

To create an object in a different location, the following questions must be answered:

- Can the client control the location for the new object?

- On the other hand, can the location be determined according to some administered policy?
- What entity does the client communicate with in order that a new object is created?
- How does the client find that entity?
- How much control does the client have over deciding the implementation of the created object?
- Can the client influence the initial values of the newly created object?
- Can the client create an object in an implementation specific fashion?

The problem of moving or copying an object

Figure 6-2 illustrates the problem of moving or copying an object in a distributed object system.

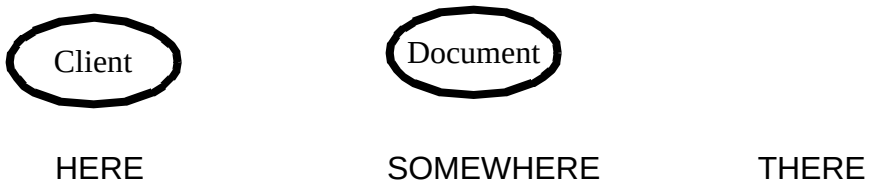

Figure6-2 Life Cycle Service defines how a client can move or copy an object over there.

To support moving or copying an object, the following questions must be answered:

- Can the client control the location for the copied or migrated object?
- On the other hand, can the location be determined according to some administered policy?
- What entity does the client communicate with to copy or migrate the object?
- How does the client find that entity?
- What happens to the implementation code of a copied or migrated object?

The problem of operating on a graph of distributed objects

Distributed objects do not *float* in space; they are connected to one another. The connections are called *relationships*. Relationships allow semantics to be added to references between objects. For example, relationships allow one object to *contain* another. Life Cycle services must work in the presence of graphs of related objects.

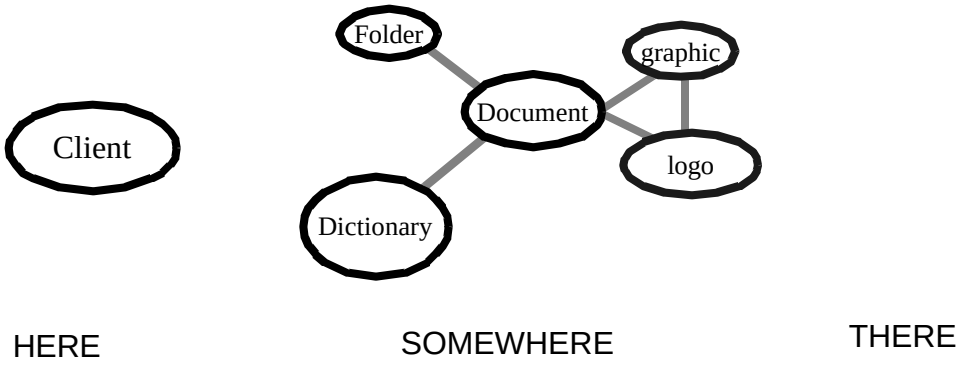

Figure6-3 The object life cycle problem for graphs of objects is to determine the boundaries of a graph of objects and operate on that graph. In the above example, a document contains a graphic and a logo, refers to a dictionary and is contained in a folder.

Figure 6-3 illustrates the object life cycle problem for graphs of objects. In the example, the folder *contains* a document, the document *contains* a graphic and a logo and *references* a dictionary. The graphic *references* the logo that is *contained* in the document. For graphs of objects, life cycle services must answer the following questions:

- What are the boundaries of the graph? For example, if a client copies the document, which objects are affected?
- If multiple objects are affected, how is the life cycle operation actually applied to those objects?
- Are cycles in the graph preserved? For example, if copying the document results in copying the graphic and the logo, is the cycle preserved in the copy?

6.1.2 Organization of this Chapter

This specification defines services and conventions to answer these life cycle issues.

Section 6.1.3 specifies a *client's model* of object life cycle. It describe the model a client has of factories and life cycle operations. A wide variety of implementations of this model are possible.

Section 6.1.4 discusses *factory finders* in detail.

Section 6.2 defines the *CosLifeCycle* module. This module defines the service interfaces and the interface supported by objects that participate in the service.

Section 6.3 discusses factory implementation strategies.

Section 6.4 discusses how objects can use factories and factory finders to support the *copy* and *move* operations.

Section 6.5 summarizes the object life cycle framework.

Appendix A contains an addendum to the Life Cycle Service; the addendum provides a specification for compound life cycle operations.

This chapter also includes additional appendices that are not part of the Life Cycle Service specification: they are included as background material. Appendix B suggests a filtering language for the filter criteria. Appendix C discusses administration of generic factories. Appendix D discusses support for PCTE objects.

6.1.3 Client's Model of Object Life Cycle

A client is any piece of code that initiates a life cycle operation for some object. A client has a simple view of the life cycle operations.

Client's Model of Creation

The client's model of creation is defined in terms of factory objects. A factory is an object that creates another object. Factories are *not* special objects. As with any object, factories have well-defined IDL interfaces and implementations in some programming language.

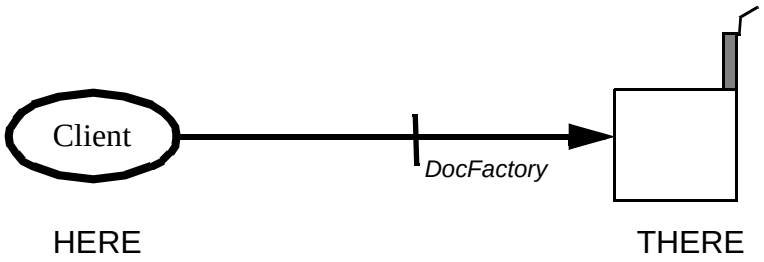

Figure6-4 To create an object "over there" a client must posses an object reference to a factory over there. The client simply issues a request on the factory.

There is no standard interface for a factory. Factories provide the client with specialized operations to create and initialize new instances in a natural way for the implementation. Figure 6-5 illustrates a factory for a document.

```
interface DocFactory {
    Document create();
    Document create_with_title(in string title);
```

```
            Document create_for(in natural_language nl);
        };
```

Figure6-5 An example of a document factory interface. This interface is defined for clients as a part of application development.

Factories are object *implementation* dependent. A different implementation of the document could define a different factory interface.

While there is no standard interface for a factory, a *generic factory* interface is defined by the life cycle service in section 6.2.3. A generic factory is a creation service. It provides a generic operation for creation. Instead of invoking an object specific operation on a factory with statically defined parameters, the client invokes a standard operation whose parameters can include information about resource filters, state initialization, policy preferences, etc.

To create an object, a client must possess an object reference for a factory, which may be either a generic factory or an object-specific factory, and issue an appropriate request on the factory. As a result, a new object is created and typically an object reference is returned.

There is nothing special about this interaction.

A factory assembles the resources necessary for the existence of an object it creates. Therefore, the factory represents a scope of resource allocation, which is the set of resources available to the factory. A factory may support an interface that enables its clients to constrain the scope.

Clients find factory objects in the same fashion they find any object. Two common scenarios for clients to find factories are:

- Clients use a finding mechanism, such a naming context, drag-and-drop, or a trader, to find factories.
- Clients are passed factory objects as a parameter to an operation the client supports.

Various *implementation* strategies for factories are discussed in detail in section 6.3.

Client's Model of Deleting an Object

A client that wishes to delete an object issues a `remove`[1] request on an object supporting the *LifeCycleObject* interface. (The *LifeCycleObject* interface is defined in section 6.2.) The object receiving the request is called the *target.*

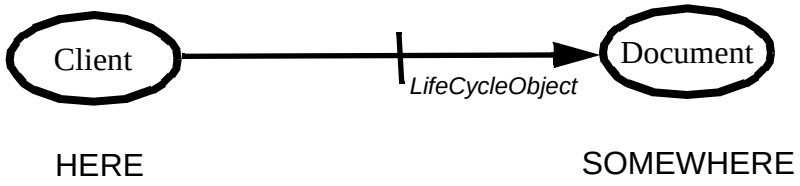

Figure6-6 To delete an object, a client must posses an object reference supporting the *LifeCycleObject* interface and issues a `remove` request on the object.

Figure 6-6 illustrates a client deleting the document.

Client's Model of Copying or Moving an Object

A client that wishes to move or copy an object issues a `move` or `copy` request on an object supporting the *LifeCycleObject* interface. The object receiving the request is called the *target.*

The move and copy operations expect an object reference supporting the *FactoryFinder* interface. The factory finder represents the "THERE" in Figure 6-7. The client is indicating to move or copy the target using a factory within the scope of the factory finder. Section 6.1.4 describes factory finders in more detail.

1. The operation is named remove, rather than delete, because delete collides with the delete operator in C++.

The implementations of move and copy can use the factory finder to find appropriate factories “over there”. Section 6.4 describes how objects can implement move and copy using the factory finder. This is invisible to the client.

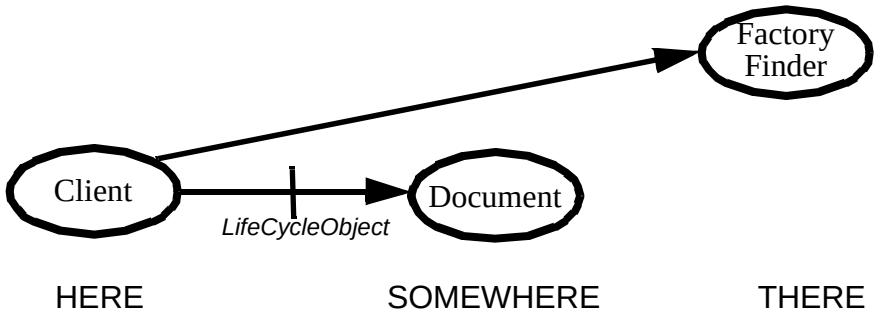

Figure6-7 Life cycle services define how a client can move or copy an object from here to there.

In the example of Figure 6-7, client code would simply issue a `copy` request on the document and pass it an object supporting the *FactoryFinder* interface as an argument.

When a client issues a copy request on a target, it is assumed that the target, the factory finder, and the newly created object can all communicate via the ORB. With externalization/internalization there is no such assumption. In the presence of a future externalization service, the externalized form of the object can exist outside of the ORB for arbitrary amounts of time, be transported by means outside of the ORB and can be internalized in a different, disconnected ORB.

Note – In general, a client is unaware of how a target and a factory finder are implemented. The target may represent a simple object or it may represent a graph of objects. Similarly, a factory finder may represent a very concrete location, such as a specific storage device, or it may represent a more abstract location, such as a group of machines. The client uses the same interface in all of these cases.

6.1.4 Factory Finders

Factory finders support an operation, `find_factories`, which returns a sequence of factories. Clients pass factory finders to the move and copy operations, which typically invoke this operation to find a factory to interact with. (This is described in detail in section 6.4.) The new copy or the migrated object will then be within the scope of the factory finder.

Some examples of locations that a factory finder might represent are:

- somewhere on a work group’s local area network
- storage device A on machine X
- Susan’s notebook computer

Multiple Factory Finders

The factory finder interface given in section 6.2 represents the minimal functionality supported by all factory finders. Target implementations can depend on this operation being available. More sophisticated factory finding facilities can be provided by extended finding services.

Currently, the only finding service being considered for standardization by the OMG is the naming service. Others are likely to be standardized in the future. It is likely that there will always be multiple finding services, of different expressive powers, in distributed object systems.

As demonstrated in Figure 6-8, the *FactoryFinder* interface can be mixed-in with interfaces for finding services, allowing multiple finding services. Many clients simply pass factory finders on to target objects. However, objects that need the services of a more powerful finding mechanism can narrow the factory finder to an appropriate, more specific interface.

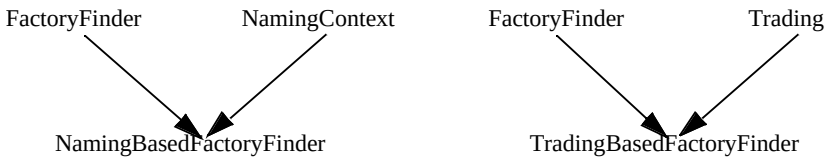

Figure6-8 The *FactoryFinder* interface can be “mixed in” with interfaces of more powerful finding services.

The power of a factory finder is determined by the power of the finding service.

6.1.5 Design Principles

Several principles have driven the design of the Life Cycle Service:

1. A factory object registered at a factory finder represents an implementation at that location. Thus, a factory finder allows clients to query a location for an implementation.
2. Object implementations can embody knowledge of finding a factory, relative to a location. Object implementations usually do not embody knowledge of location.
3. The desired result for life cycle operations such as copy and move depends on relationships between the target object and other objects. The design given in Appendix A has built-in support for the two most basic kinds of relationships, *containment* and *reference*, and supports the definition of new kinds of relationships and propagation semantics.
4. The Life Cycle Service is not dependent on any particular model of persistence and is suitable for distributed, heterogeneous environments.
5. The design does not include an object equivalence service nor rely on global object identifiers.

6.1.6 Resolution of Technical Issues

This specification addresses the following issues that were identified for the Life Cycle Service in the OMG *Object Services Architecture*[2] :

- *Creation*: Many of the parameters supplied to an object `create` operator will be implementation-dependent, so that a standardized universal IDL signature for object creation is not possible. IDL signatures for object creation will be defined for various kinds of object factories, but the signatures will be specific to type, implementation, and persistent storage mechanism of the object to be created.
- *Deletion*: A `remove` operator is defined on any object supporting the *LifeCycleObject* interface. This model for deletion supports any desired paradigm for referential integrity. Appendix A describes support for the two most common paradigms, based on reference and containment relationships. Only one type of deletion is supported; a different operation should be used for archiving an object. This interface can support many paradigms for storage management, e.g. garbage collection and reference counts. Since storage management is implementation-dependent, its interface does not belong in the generalized life cycle interfaces.
- *Copying*: Appendix A describes support for shallow and deep copy, and referential integrity. A scheme based on reference and containment relationships defines scopes for operations such as copy. The concept of an *factory finder* is used for object location. This paradigm for copying, deleting, and moving objects works regardless of an object's ORB, persistent storage mechanism, and implementation. This design is extensible because objects participate in the traversal algorithm, and the relationship service presented in the appendix supports the definition of new kinds of relationships with different behavior.
- *Equivalence*: There was no need for an object equivalence service or global object identifiers in the design of the Life Cycle Service to support real world applications or other object services.

2. *Object Services Architecture*, Document Number 92-8-4, Object Managment Group, Framingham, MA, 1992.

6.2 The CosLifeCycle Module

Client code accesses the basic life cycle functionality via the *CosLifeCycle* module. This module defines the *FactoryFinder*, *LifeCycleObject* and *GenericFactory* interfaces and describes the operations of these interfaces in detail.

```
#include “Naming.idl”

module CosLifeCycle{

    typedef Naming::Name Key;
    typedef Object Factory;
    typedef sequence <Factory> Factories;
    typedef struct NVP {
        Naming::Istring name;
        any value;
    } NameValuePair;
    typedef sequence <NameValuePair> Criteria;

    exception NoFactory {
        Key search_key;
    };
    exception NotCopyable { string reason; };
    exception NotMovable { string reason; };
    exception NotRemovable { string reason; };
    exception InvalidCriteria{
        Criteria invalid_criteria;
    };
    exception CannotMeetCriteria {
        Criteria unmet_criteria;
    };
```

Figure6-9 The CosLifeCycle Module

```
    interface FactoryFinder {
        Factories find_factories(in Key factory_key)
            raises(NoFactory);
    };

    interface LifeCycleObject {
        LifeCycleObject copy(in FactoryFinder there,
                in Criteria the_criteria)
            raises(NoFactory, NotCopyable, InvalidCriteria,
                 CannotMeetCriteria);
        void move(in FactoryFinder there,
                in Criteria the_criteria)
            raises(NoFactory, NotMovable, InvalidCriteria,
                 CannotMeetCriteria);
        void remove()
            raises(NotRemovable);
    };

    interface GenericFactory {
        boolean supports(in Key k);
        Object create_object(
                in Key     k,
                in Criteria the_criteria)
            raises (NoFactory, InvalidCriteria,
                CannotMeetCriteria);
    };
};
```

Figure6-9 The CosLifeCycle Module

6.2.1 The LifeCycleObject Interface

The *LifeCycleObject* interface defines `copy`, `move` and `remove` operations. Objects participate in the life cycle service by supporting this interface.

copy

```
    LifeCycleObject copy(in FactoryFinder there,
            in Criteria the_criteria)
        raises(NoFactory, NotCopyable, InvalidCriteria,
            CannotMeetCriteria);
```

The `copy` operation makes a copy of the object. The copy is located in the scope of the factory finder passed as the first parameter. The copy operation returns an object reference to the new object. The new object is initialized from the existing object.

The first parameter, `there`, may be a nil object reference. If passed a nil object reference, the target object can determine the location or fail with the `NoFactory` exception.

The second parameter, `the_criteria`, allows for a number of optional parameters to be passed. Typically, the target simply passes this parameter to the factory used in creating the new object. The criteria parameter is explained in detail in section 6.2.4

If the target cannot find an appropriate factory to create a copy “over there”, the `NoFactory` exception is raised. An implementation that refuses to copy itself should raise the `NotCopyable` exception. If the target does not understand the criteria, the `InvalidCriteria` exception is raised. If the target understands the criteria but cannot satisfy the criteria, the `CannotMeetCriteria` exception is raised.

In addition to these exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the copied object, `NO_RESOURCES` will be raised. Similarly, if a target does not implement the copy operation, the `NO_IMPLEMENT` exception will be raised.

It is implementation dependent whether this operation is *atomic*.

move

The `move` operation on the target moves the object to the scope of the factory finder passed as the first parameter. The object reference for the target object remains valid after `move` has successfully executed.

The first parameter, `there`, may be a nil object reference. If passed a nil object reference, the target object can determine the location or fail with the `NoFactory` exception.

The second parameter, `the_criteria`, allows for a number of optional parameters to be passed. Typically, the target simply passes this parameter to the factory used in migrating the new object. The criteria parameter is explained in detail in section 6.2.4

If the target cannot find an appropriate factory to support migration of the object “over there”, the `NoFactory` exception is raised. An implementation that refuses to move itself should raise the `NotMovable` exception. If the target does not understand the criteria, the `InvalidCriteria` exception is raised. If the target understands the criteria but cannot satisfy the criteria, the `CannotMeetCriteria` exception is raised.

In addition to these exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for migrating the object, `NO_RESOURCES` will be raised. Similarly, if a target does not implement the move operation, the `NO_IMPLEMENT` exception will be raised.

remove

```
void remove()
    raises(NotRemovable);
```

`Remove` instructs the object to cease to exist. The object reference for the target is no longer valid after remove successfully completes. The client is not responsible for cleaning up any resources the object uses. An implementation that refuses to remove itself should raise the `NotRemovable` exception. In addition to this exception, implementations may raise standard CORBA exceptions.

6.2.2 The FactoryFinder Interface

Factory finders support an operation, `find_factories`, which returns a sequence of factories. Clients pass factory finders to the move and copy operations, which typically invoke this operation to find a factory to interact with. (This is described in detail in section 6.4.)

The factory finder interface represents the *minimal* functionality supported by all factory finders.

find_factories

```
Factories find_factories(in Key factory_key)
    raises(NoFactory);
```

The `find_factories` operation is passed a key used to identify the desired factory. The key is a name, as defined by the naming service. More than one factory may match the key. As such, the factory finder returns a sequence of factories. If there are no matches, the `NoFactory` exception is raised.

The scope of the key is the factory finder. The factory finder assigns no semantics to the key. It simply matches keys. It makes no guarantees about the interface or implementation of the returned factories or objects they create.

It is beyond the scope of this specification to standardize the key space. The space of keys is established by *convention* in particular environments. The *kind* field[3] of the key is useful for partitioning the key space. Suggested values for the `id` and `kind` fields are given in Table 6-1.

Table 6-1 Suggested conventions for factory finder keys.

id field	**kind field**	**meaning**
name of object interface	“object interface”	Find factories that create objects supporting the named interface.
name of equivalent implementations	“implementation equivalence class”	Find factories that create objects with implementations in a named equivalence class of implementations.[1]
name of object implementation	“object implementation”	Find factories that create objects of a particular implementation.
name of factory interface	“factory interface”	Find factories supporting the named factory interface.

1. An example of an implementation equivalence class is a set of object implementations that have compatible externalized forms.

6.2.3 The GenericFactory Interface

In many environments, management of a set of resources that are allocated to objects at creation time is required. This needs to be done in a coordinated fashion for all types of objects. The Life Cycle Service provides a framework for this which is intended to be usable in a variety of administrative environments. However, the differing environments will administer a variety of resources and it is beyond the scope of this framework to identify all the possible types of resource.

While there is no standard interface for a factory, a *GenericFactory* interface is defined. The *GenericFactory* interface defines a generic creation operation, `create_object`. By defining a generic interface for creation, a creation service can be implemented. This is particularly useful in environments where administering a set of resources is important.

Such a generic factory can implement resource policies and represent multiple locations. In administered environments, object specific factories, such as the document factory described in section , may delegate the creation process to the generic factory. This is described in detail in section 6.3.2.

The job of the generic factory is to match the creation criteria specified by clients of the *GenericFactory* interface with offers made on behalf of implementation specific factories.

3. See the naming service specification.

Figure 6-10 illustrates the structure of a creation service.

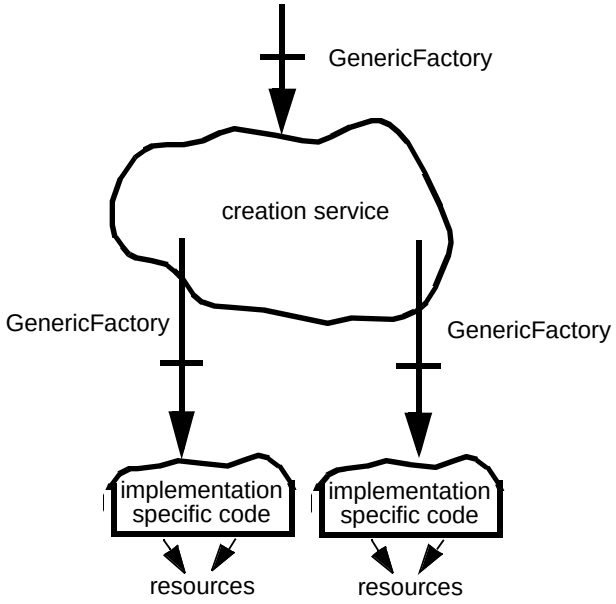

Figure6-10 The Life Cycle service provides a generic creation capability. Ultimately, implementation specific creation code is invoked by the creation service. The implementation specific code also supports the *GenericFactory* interface.

The client of the *GenericFactory* interface invokes the `create_object` operation and can express criteria for creation.

Ultimately, this request will be passed to an implementation specific factory which supports the *GenericFactory* interface. To get there, the request may travel through a number of generic factories. However, all of this is transparent to the client.

create_object

```
Object create_object(
        in Key k,
        in Criteria the_criteria)
    raises (NoFactory, InvalidCriteria,
        CannotMeetCriteria);
```

The `create_object` operation is passed a key used to identify the desired object to be created. The key is a name, as defined by the Naming Service.

The scope of the key is the generic factory. The generic factory assigns no semantics to the key. It simply matches keys. It makes no guarantees about the interface or implementation of the created object.

It is beyond the scope of this specification to standardize the key space. The space of keys is established by *convention* in particular environments. The *kind* field[4] of the key is useful for partitioning the key space. *Suggested* values for the `id` and `kind` fields are given in Table 6-2.

Table 6-2 Suggested conventions for generic factory keys.

id field	kind field	meaning
name of object interface	“object interface”	Create an object that supports the named interface.
name of equivalent implementations	“implementation equivalence class”	Create an object whose implementation is in a named equivalence class of implementations.[1]
name of object implementation	“object implementation”	Create objects of a particular implementation.

1. An example of an implementation equivalence class is a set of object implementations that have compatible externalized forms

The second parameter, `the_criteria`, allows for a number of optional parameters to be passed. Criteria are explained in detail in section 6.2.4

If the generic factory cannot create an object specified by the key, then `NoFactory` is raised.

If the target does not understand the criteria, the `InvalidCriteria` exception is raised. If the target understands the criteria but cannot satisfy the criteria, the `CannotMeetCriteria` exception is raised.

supports

```
boolean supports(in Key k);
```

The `supports` operation returns `true` if the generic factory can create an object, given the key. Otherwise `false` is returned.

4. See the naming service specification.

6.2.4 Criteria

The `create_object` operation of the *GenericFactory* interface expects a parameter specifying the creation criteria. The `move` and `copy` operations of the *LifeCycleObject* interface also expects this parameter; typically they pass it through to a factory. This section documents this parameter.

The criteria parameter is expressed as an IDL sequence of name-value pairs. In particular, it is described by the following data structure given in the *CosLifeCycle* module:

The parameter is given as a sequence of name-value pairs in order to be extensible and support "pass-through"; that is, new name-value pairs can be defined in the future and objects can be written that do not interpret the name-value pairs, but just pass them on to other objects.

Note – It is beyond the scope of this specification to standardize particular criteria. Supporting criteria is optional. Furthermore, supporting different criteria is acceptable. The criteria given here are suggestions.

Table 6-3 suggests criteria to be supported by the generic factory. Detailed descriptions follow.

"initialization"

The "initialization" criterion is a sequence of name-value pairs which is intended to contain application specific initialization values. Typically, the generic factory will pay no attention to the initialization criterion and simply passes it on to application specific factory code.

"filter"

The filter criterion is a constraint expression which provides the client with a powerful way of expressing its requirements on creation. The generic factory will use the constraint expression to make decisions about the allocation of particular resources. For example, a client could give a constraint "operating system" != "windows nt".

These constraints are expressed in some Constraint Language. A constraint language is suggested in Appendix B.

Filters are potentially complex and `InvalidCriteria` will be raised if the filter is too complex for the factory or is syntactically incorrect.

"logical location"

The "logical location" criterion allows a client to express where a created/copied/migrated object is logically created. For example, in PCTE an object is always in a relationship with another object. In such an environment, the logical location would specify another object and a relationship.

"preferences"

The "preferences" criterion allows the client to influence the policies which the generic factory uses to make decisions. For example, a generic factory might arbitrarily choose a machine from a set of machines. Using the preferences criterion, a client could express its preference for a particular machine. Policies and preferences are described in more detail in Appendix B.

6.3 Implementing Factories

As defined under Client's Model of Creation on page 4, any object that creates another object in response to some request is called a *factory*. Clients depend only on the definitions in that section.

The client's model of object life cycle has intentionally been defined abstractly. This allows a wide variety of implementation strategies.

Factories are *not* special objects. They have well-defined IDL interfaces and implementations in programming languages. Defining factory interfaces and implementing them are a normal part of application development.

Ultimately, the creation process requires implementation dependent code that assembles resources for the storage and execution of an object. The act of creating an object requires assembling and initializing all of the resources required to support the execution and storage of the object. The resources typically include:

- the allocation of one or more BOA object references, and
- resources related to persistence storage.

6.3.1 Minimal Factories

Figure 6-11 illustrates a minimal implementation of a factory that assembles resources in a single factory object.

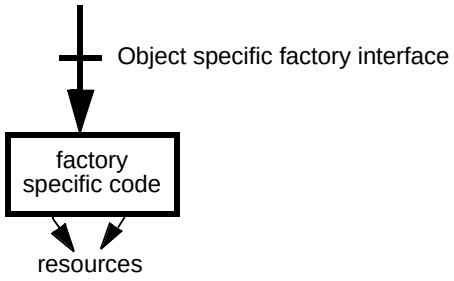

Figure6-11 Factories assemble resources for the execution of an object. A minimal implementation achieves this with a single factory implementation.

6.3.2 Administered Factories

Factories can delegate the creation process to a generic factory that administers a set of resources. The generic factory may apply policies to all creation requests.

Eventually such a generic creation service, needs to communicate with implementation specific code that actually assembles the resources for the object. Figure 6-12 illustrates an object specific factory, such as the document factory of Figure 6-5 that delegates the creation problem to the generic creation service. The object-specific factory effectively adds a statically typed wrapper around the generic factory.

.

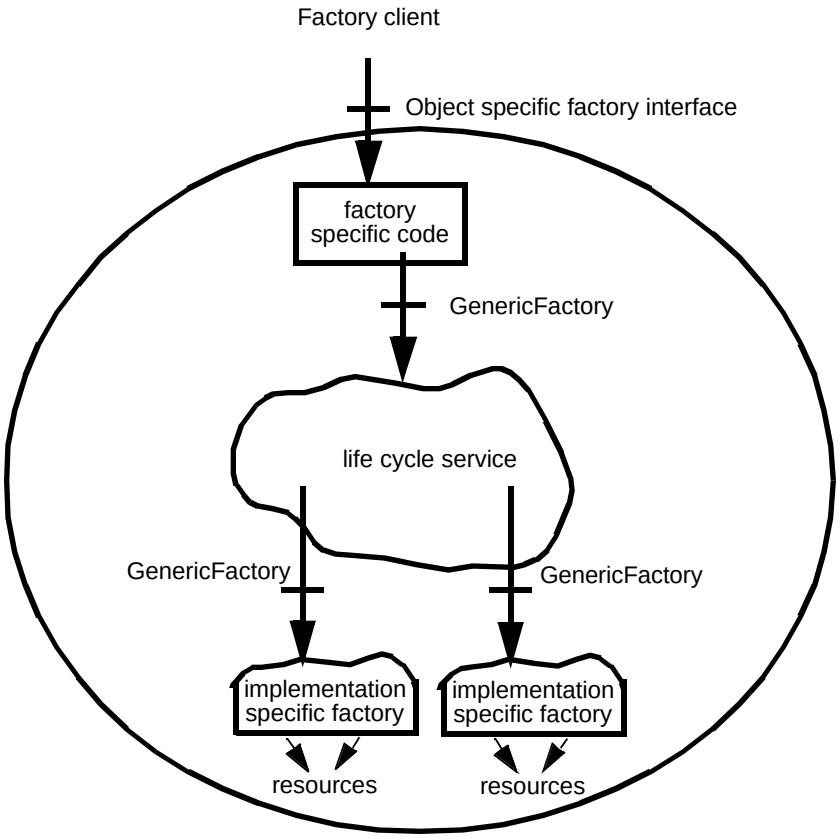

Figure6-12 In an administered environment, factory *implementations* can delegate the creation problem to a generic factory. The generic factory can apply resource allocation policies. Ultimately the creation service communicates with implementation specific code that assembles resources for the object.

6.4 Target's Use of Factories and Factory Finders

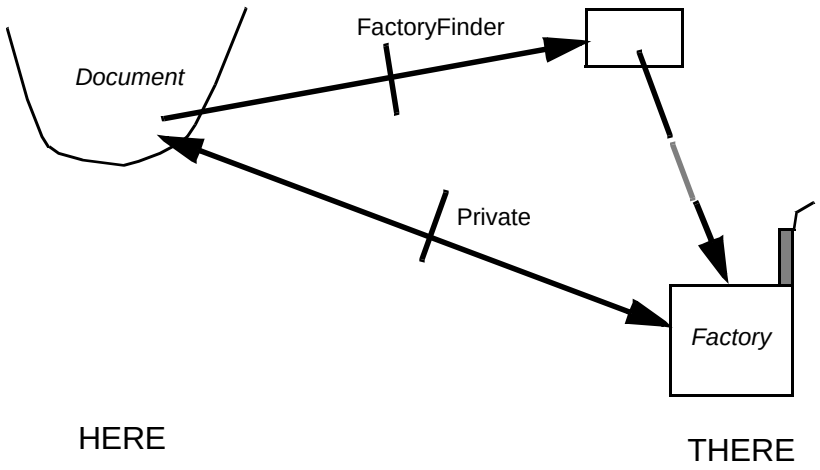

Figure6-13 The copy and move operations are passed a *FactoryFinder* to represent "there." The implementation of the target uses the *FactoryFinder* to find a factory object for creation over there. The protocol between the object and the factory is private. They can communicate and transfer state according to any implementation-defined protocol.

A client passes a factory finder as a parameter to a copy or move request.

Clients do not generally understand the implementation constraints of the object being copied. Clients cannot express what the target object needs in order to copy itself to the new location.

Target object implementations, on the other hand, put constraints on factories based on implementation concerns. It is unlikely that target implementation code is interested in further constraining location.

To find an appropriate factory, the target object implementation may use the factory finder with its minimal interface defined in section 6.2.2 or it may attempt to narrow the factory finder to a more sophisticated finding service with more expressive power. The target object implementation can always depend on the existence of the minimal interface.

Once the target object implementation finds a factory, it communicates with the factory using a private, implementation-defined, interface.

6.5 Summary of Life Cycle Service

The problem of distributed object life cycle is the problem of

- Creating an object
- Deleting an object

- Moving and copying an object
- Operating on a graph of distributed objects.

The client's model of object life cycle is based on *factories* and target objects supporting the *LifeCycleObject* interface. Factories are objects that create other objects. The *LifeCycleObject* interface defines operations to delete an object, to move an object and to copy an object.

A *GenericFactory* interface is defined. The generic factory interface is sufficient to create objects of different types. By defining a *GenericFactory* interface, implementations that administer resources are enabled.

6.5.1 Summary of Life Cycle Service Structure

The Life Cycle Service specification consists of these interfaces:

- *LifeCycleObject*
- *FactoryFinder*
- *GenericFactory*
- Interfaces described in Appendix A, an addendum to the Life Cycle Service

Appendix A Addendum to Life Cycle Service: Compound Life Cycle Specification

This appendix contains the specification for the compound life cycle component of the Life Cycle Service .The compound life cycle specification depends on the Life Cycle Service for the definition of the client view of Life Cycle operations. Moreover, it extends the Life Cycle Service to support compound life cycle operations on graphs of related objects. In addition, the compound life cycle specification depends on the Relationship Service for the definition of object graphs.

The Life Cycle Service specification describes a client's view of object life cycle. It describes how a client can *c*reate, copy, move and remove objects in a distributed object system. To create objects, clients find *factory objects* and issue create requests on factories. To copy, move and remove objects, clients issue requests on target objects supporting the *LifeCycleObject* interface.

If the target object represents a simple object, that is an object that is not part of a graph of related objects, the target provides an implementation for each of the operations in the *LifeCycleObject* interface.

If, on the other hand, the target object uses the Relationship Service for representing relationships with other objects, additional services are available to implement the compound life cycle operations. The specification in this appendix describes those services.

A.1 Key Features

The compound life cycle specification:

- Addresses the issues of copying, moving and removing objects that are related to other objects. Depending on the semantics of the relationships, these life cycle operations are applied to:
- the object, to the relationship and to the related objects
- the object and to the relationship
- the object
- Coordinates compound life cycle operations on graphs of related objects, thus relieving object developers from implementing compound operations.
- Illustrates a general model for applying compound operations to graphs of related objects. The Externalization Service also illustrates the model.

A.2 Service Structure

The specification in this appendix defines a service that applies a compound life cycle operation to a graph of related objects, given a starting node. Compound operations traverse a graph of related objects and apply the operation to the relevant nodes, roles and relationships of the graph. The service supports the *CosCompoundLifeCycle::Operations* interface. Implementations of the service depend on the *CosCompoundLifeCycle::Node* , *CosCompoundLifeCycle::Role* and *CosCompoundLifeCycle::Relationship* interfaces which are subtypes of the *Node* , *Role*

and *Relationship* interfaces defined in the Relationship Service. The *CosCompoundLifeCycle::Node* , *CosCompoundLifeCycle::Role* and *CosCompoundLifeCycle::Relationship* interfaces add operations to copy, remove and move nodes, roles and relationships.

The Relationship Service defines interfaces for containment and reference relationships and their roles. This appendix defines interfaces that inherit those interfaces and the compound life cycle interfaces.

A.3 Interface Overview

Table 6-4 and Table 6-5 summarize the interfaces defined in the *CosCompoundLifeCycle* module. The *CosCompoundLifeCycle* module is described in detail in sectionSection A.4.2.

Table 6-4 Interfaces defined in the *CosCompoundLifeCycle* module for initiating compound life cycle operations.

Interface	Purpose
Operations	Defines compound life cycle operations on graphs of related objects.
OperationsFactory	Defines an operation to create an object that supports the *Operations* interface.

Table 6-5 Interfaces defined in the *CosCompoundLifeCycle* module that are used by implementations of compound life cycle operations

Interface	Inherits	Purpose
Node	CosGraphs::Node	Defines life cycle operations on nodes in graphs of related objects.
Relationship	CosRelationships::Relationship	Defines life cycle operations on relationships.
Role	CosGraphs::Role	Defines life cycle operations on roles.
PropagationCriteriaFactory		Creates an object that supports the CosGraphs::TraversalCriteria interface that uses relationship propagation values.

Table 6-6 and Table 6-7 summarize the interfaces that combine the specific relationships defined by the Relationship Service and the life cycle interfaces defined in this appendix.

Table 6-6 Interfaces defined in the *CosLifeCycleContainment* module.

Interface	Inherits	Purpose
Relationship	CosContainment::Containment and CosCompoundLifeCycle::Relationship	Combines both interfaces. No additional operations are defined.
ContainsRole	CosContainment::ContainsRole and CosCompoundLifeCycle::Role	Combines both interfaces. No additional operations are defined.
ContainedInRole	CosContainment::ContainedInRole and CosCompoundLifeCycle::Role	Combines both interfaces. No additional operations are defined.

Table 6-7 Interfaces defined in the *CosLifeCycleReference* module.

Interface	Inherits	Purpose
Relationship	CosContainment::Reference and CosCompoundLifeCycle::Relationship	Combines both interfaces. No additional operations are defined.
ReferencesRole	CosContainment::ReferencesRole and CosCompoundLifeCycle::Role	Combines both interfaces. No additional operations are defined.
ReferencedByRole	CosContainment::ReferencedByRole and CosCompoundLifeCycle::Role	Combines both interfaces. No additional operations are defined.

A.4 Compound Life Cycle Operations

The Life Cycle specification describes a client's view of object life cycle. It describes how a client can *create*, *copy*, *move* and *remove* objects in a distributed object system. To create objects, clients find *factory objects* and issue create requests on factories. To copy, move and remove objects, clients issue requests on target objects supporting the *LifeCycleObject* interface.

If the target object represents a simple object, that is an object that is not part of a *graph of related* objects, the target provides an implementation for each of the operations in the *LifeCycleObject* interface.

If the target participates as a node in a graph of related objects, the target can delegate the life cycle operation to a service that implements the compound life cycle operation. In particular, the target simply creates an object that supports the *CosCompoundLifeCycle::Operations* interface and issues the corresponding life cycle request on it. The compound life cycle operations expect a *CompoundLifeCycle::Node* object reference as a starting node. The target simply passes its *CompoundLifeCycle::Node* object reference as the starting node.

When the life cycle object has completed issuing compound life cycle requests, it simply issues the `destroy` request to destroy the compound operation.

Figure 6-14 illustrates the target's delegation of the life cycle request to compound operation.

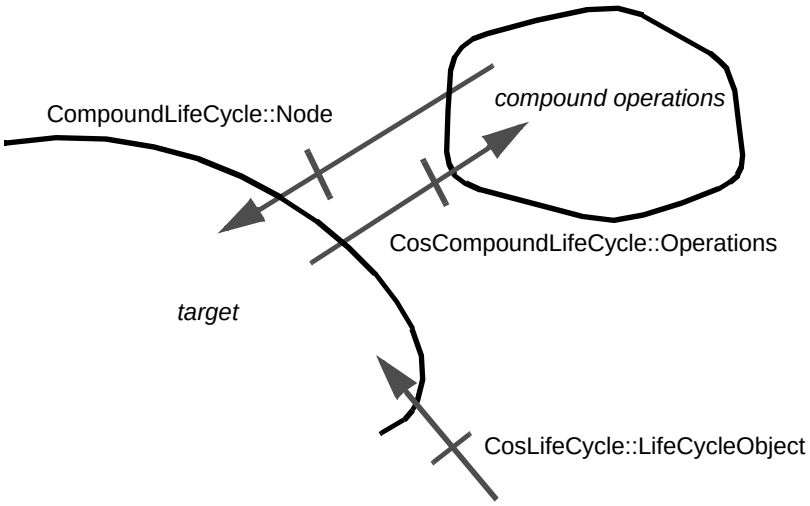

Figure 6-14 A life cycle object that is part of a graph of related objects delegates the orderly operation on the graph to an object that implements the compound life cycle operation.

A.4.1 Applying the Copy Operation to the Example

We now use the example in the Relationship Service Specification (Figure 9-3) to illustrate applying the copy operation to a graph. Figure 6-15 illustrates the graph and the compound operation prior to applying the copy operation. Recall that the folder *contains* the document; the document is *contained in* the folder. The document *contains* the figure; the figure is *contained in* the document. The document *contains* the logo and the logo is *contained in* the document. On the other hand, the document *references* the book; the book is *referenced by* the document. Finally, the figure *references* the logo; the logo is *referenced by* the figure.

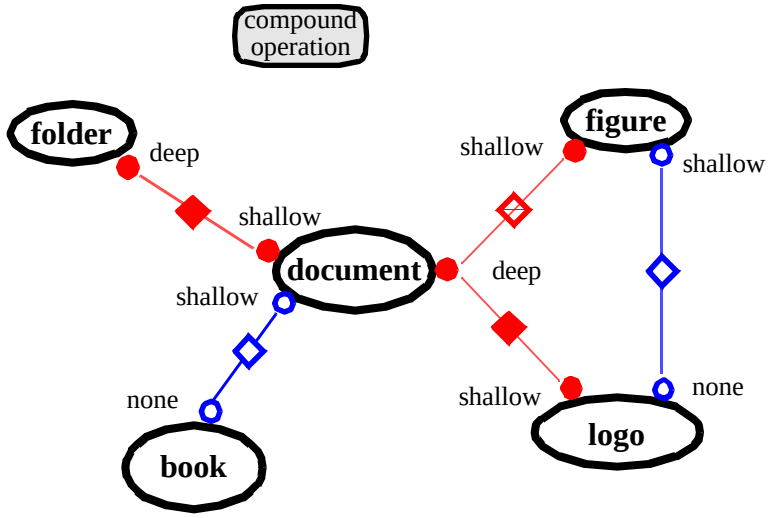

Figure6-15 Prior to applying copy to the graph.

In this example, the copy is performed in two passes. The first pass creates a list representation of the relevant edges of the graph. The second pass takes the list as input, copies the relevant nodes and roles, then creates all the necessary links by copying the relevant relationships.

A compound copy request is initiated by issuing a *LifeCycleObject*`t::copy` request on the folder. Since the folder participates in a graph of related objects, it creates an object supporting the *CosCompoundLifeCycle::Operations* interface (the *Operations* object). Then the folder issues a *CosCompoundLifeCycle::Operations*::`copy` request on the *Operations* object, passing in its own *CosCompoundLifeCycle::Node* object reference as the starting node. The copy operation will copy the graph of related objects and return an object reference for the copy of the folder object.

The remainder of this section provides a description of how the *Operations* object might implement the copy operation.

First Pass of the Compound Copy Operation

The first pass consists of creating a list representation of the relevant edges of the graph. The *Operations* object uses an object supporting the *CosGraphs::Traversal* interface to do most of the work.

The *Operations* object creates an object supporting the *CosGraphs::TraversalCriteria* interface by calling *CosCompoundLifeCycle::PropagationCriteriaFactory*::`create`.

The *Operations* object then creates a *CosGraphs::Traversal* object by calling *CosGraphs::TraversalFactory*::`create_traversal_on`, passing in the object supporting the *CosGraphs::TraversalCriteria* interface. Calls on the *CosGraphs::Traversal* object yield an unordered list of *CosGraphs::Traversal*::*ScopedEdges* containing the following information.

(folder, ContainsRole, Containment, ContainedInRole, document)
(document, ReferencesRole, Reference, ReferencedByRole, book)
(document, ContainedInRole, Containment, ContainsRole, folder)
(document, ContainsRole, Containment, ContainedInRole, figure)
(document, ContainsRole, Containment, ContainedInRole, logo)
(figure, ReferencesRole, Reference, ReferencedByRole, logo)
(figure, ContainedInRole, Containment, ContainsRole, document)
(logo, ContainedInRole, Containment, ContainsRole, document)

This list will be referred to as the *OriginalEdgeList*.

Since the propagation value for copy from the document to the book is shallow, the traversal did not visit the book. As such, the edge:

(book, ReferencedByRole, Reference, References, document)

is not included. Although the traversal did visit the logo, the edge

(logo, ReferencedByRole, Reference, ReferencesRole, figure)

is not included because the propagation value for copy from the logo to the figure is none.

For more detailed information regarding the output of the *CosGraphs::Traversal* object with respect to the use of propagation semantics, see section 9.4.3 of the Relationship Service.

Second Pass of the Compound Copy Operation

The second pass copies all the relevant nodes and then relates them by copying the relevant relationships.

First, the set of nodes to be copied must be determined. This consists of all the distinct nodes in the left column of the *OriginalEdgeList* . Since a node may be involved in multiple edges, it may appear multiple times in the list; it should only be copied once. Each node in this set is copied by issuing a *CosCompoundLifeCycle::Node*::`copy_node` request. This request will cause the node *and all of its roles* to be copied; the new node and its roles will be returned.

- For each returned role of the copied node, an entry is made in a table of new roles. Each entry consists of:
- The role object is the data and
- The node's CosGraphs::Traversal::TraversalScopedId and the role's CORBA::InterfaceDef together serve as a key.

The final step is to create all the relationships for the copied graph. All of the distinct relationships in the center column of the *OriginalEdgeList* need to be copied. Although a relationship may appear multiple times in the list, it should only be copied once. Each relationship is copied by issuing a *CosCompoundLifeCycle::Relationship*::`copy_relationship` request. The arguments to *CosCompoundLifeCycle::Relationship*::`copy_relationship` include the list of roles to be included in the new relationship. Some of these roles will be copies that were created as a result of processing deep propagation values; others will be roles in the original graph.

Thus, copy each unique relationship in the *OriginalEdgeList*, using NamedRoles as follows:

For each role in an entry in the *OriginalEdgeList*, make a role key using the node's TraversalScopedId and the role's CORBA::InterfaceDef to search the table of new roles.

a. If the role was copied, the key will find the role's copy. The role's *RoleName* is obtained from the entry in the *OriginalEdgeList*. The role's copy and the *RoleName* are combined to form a *CosGraphs::NamedRole* which will then be included in the list of *CosGraphs::NamedRole*s passed to the *CosCompoundLifeCycle::Relationship*::`copy_relationship` method.

b. If no copy is found, the original *CosGraphs::NamedRole* is used instead.

Once all the *Relationships* have been copied, the *CosCompoundLifeCycle::Operations*::`copy` method is done.

Figure 6-16 illustrates the result of applying copy to the graph, starting at the folder.

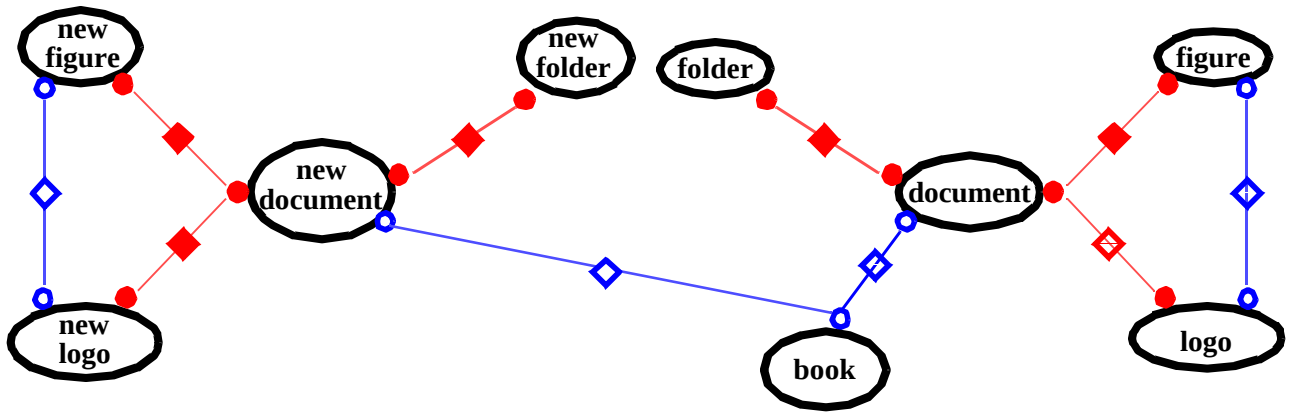

Figure6-16 The result of applying copy to the graph, starting at the folder.

When the copy operation propagates to a node because of a deep propagation value, other shallow propagation values to that node are *promoted*. That is, they are processed as if they were deep; relationships are formed with the copied node, not with the

original. This happened in the example; the shallow propagation value from the figure to the logo was promoted to deep because the logo was copied. As such, the new figure references the new logo, *not* the original logo.

A.4.2 The CosCompoundLifeCycle Module

The *CosCompoundLifeCycle* module defines

- The *Operations* interface for initiating compound life cycle operations on graphs of related objects,
- *OperationsFactory* interface for creating compound operations,
- The *Node*, *Role*, *Relationship* and *PropagationCriteriaFactory* interfaces for use by implementations of compound life cycle operations.

The *CosCompoundLifeCycle* module is given in Figure 6-17. Detailed descriptions of the interfaces follow.

```
#include <LifeCycle.idl>
#include <Relationships.idl>
#include <Graphs.idl>

module CosCompoundLifeCycle {
    interface OperationsFactory;
    interface Operations;
    interface Node;
    interface Role;
    interface Relationship;
    interface PropagationCriteriaFactory;

    enum Operation {copy, move, remove};

    struct RelationshipHandle {
        Relationship the_relationship;
        ::CosObjectIdentity::ObjectIdentifier constant_random_id;
    };

    interface OperationsFactory {
        Operations create_compound_operations();
    };
```

Figure6-17 The CosCompoundLifeCycle Module

```
interface Operations {
    Node copy (
            in Node starting_node,
            in ::CosLifeCycle::FactoryFinder there,
            in ::CosLifeCycle::Criteria the_criteria)
        raises (::CosLifeCycle::NoFactory,
            ::CosLifeCycle::NotCopyable,
            ::CosLifeCycle::InvalidCriteria,
            ::CosLifeCycle::CannotMeetCriteria);
    void move (
            in Node starting_node,
            in ::CosLifeCycle::FactoryFinder there,
            in ::CosLifeCycle::Criteria the_criteria)
        raises (::CosLifeCycle::NoFactory,
            ::CosLifeCycle::NotMovable,
            ::CosLifeCycle::InvalidCriteria,
            ::CosLifeCycle::CannotMeetCriteria);
    void remove (in Node starting_node)
        raises (::CosLifeCycle::NotRemovable);
    void destroy();
};

interface Node : ::CosGraphs::Node {
    exception NotLifeCycleObject {};
    void copy_node ( in ::CosLifeCycle::FactoryFinder there,
            in ::CosLifeCycle::Criteria the_criteria,
            out Node new_node,
            out Roles roles_of_new_node)
        raises (::CosLifeCycle::NoFactory,
            ::CosLifeCycle::NotCopyable,
            ::CosLifeCycle::InvalidCriteria,
            ::CosLifeCycle::CannotMeetCriteria);
    void move_node (in ::CosLifeCycle::FactoryFinder there,
            in ::CosLifeCycle::Criteria the_criteria)
        raises (::CosLifeCycle::NoFactory,
            ::CosLifeCycle::NotMovable,
            ::CosLifeCycle::InvalidCriteria,
            ::CosLifeCycle::CannotMeetCriteria);
    void remove_node ()
        raises (::CosLifeCycle::NotRemovable);
    ::CosLifeCycle::LifeCycleObject get_life_cycle_object()
        raises (NotLifeCycleObject);
};
```

Figure6-17 The CosCompoundLifeCycle Module *(Continued)*

```
interface Role : ::CosGraphs::Role {
```

A.4.3 The OperationsFactory Interface

Creating a Compound Life Cycle Operation

```
Operations create_compound_operations();
```

The `create_compound_operations` operation creates an object that implements the compound life cycle operations, that is, the factory creates and returns an object that supports the *CosCompoundLifeCycyle::Operations* interface.

The Operations Interface

The *Operations* interface defines compound life cycle operations to copy, move and remove objects, given a starting node in a graph.

Applying the Copy Operation to a Graph of Related Objects

```
Node copy (
        in Node starting_node,
        in ::CosLifeCycle::FactoryFinder there,
        in ::CosLifeCycle::Criteria the_criteria)
    raises (::CosLifeCycle::NoFactory,
            ::CosLifeCycle::NotCopyable,
            ::CosLifeCycle::InvalidCriteria,
            ::CosLifeCycle::CannotMeetCriteria);
```

The `copy` operation applies the copy operation to a graph of related objects. The starting node is provided as the `starting_node` parameter. The copy should be collocated with the factory finder given by the `there` parameter. The final parameter, `the_criteria`, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

If a node, role or relationship in the graph refuses to be copied, the `NotCopyable` exception is raised with the node, role or relationship object reference returned as a parameter to the exception.

If appropriate factories to create a copies of the nodes and roles cannot be found, the `NoFactory` exception is raised. The exception value indicates the key used to find the factory.

In addition to the `NoFactory` and `NotCopyable` exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the copied graph, `NO_RESOURCES` will be raised.

It is implementation dependent whether this operation is *atomic*.

Applying the Move Operation to a Graph of Related Objects

```
void move (
        in Node starting_node,
        in ::CosLifeCycle::FactoryFinder there,
        in ::CosLifeCycle::Criteria the_criteria)
    raises (::CosLifeCycle::NoFactory,
        ::CosLifeCycle::NotMovable,
        ::CosLifeCycle::InvalidCriteria,
        ::CosLifeCycle::CannotMeetCriteria);
```

The `move` operation applies the move operation to a graph of related objects. The starting node is provided as the `starting_node` parameter. The migrated graph should be collocated with the factory finder given by the `there` parameter. The final parameter, `the_criteria`, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

If a node, role or relationship in the graph refuses to be moved, the `NotMovable` exception is raised with the node, role or relationship object reference returned as a parameter to the exception.

If appropriate factories to migrate the nodes and roles cannot be found, the `NoFactory` exception is raised. The exception value indicates the key used to find the factory.

In addition to the `NoFactory` and `NotMovable` exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the migrated graph, `NO_RESOURCES` will be raised.

It is implementation-dependent whether this operation is *atomic*.

Applying the Remove Operation to a Graph of Related Objects

```
void remove (in Node starting_node)
    raises (::CosLifeCycle::NotRemovable);
```

The `remove` operation applies the remove operation to a graph of related objects. The starting node is provided as the `starting_node` parameter.

If a node, role or relationship in the graph refuses to be removed, the `NotRemovable` exception is raised with the node, role or relationship object reference returned as a parameter to the exception.

It is implementation dependent whether this operation is *atomic*.

Destroying the Compound Operation

```
void destroy();
```

The destroy operation indicates to the compound operation that the client has completed operating on the graph. The compound operation object is destroyed.

The Node Interface

The *Node* interface defines operations to copy, move and remove a node.

Copying a Node

```
void copy_node ( in ::CosLifeCycle::FactoryFinder there,
        in ::CosLifeCycle::Criteria the_criteria,
        out Node new_node,
        out Roles roles_of_new_node)
    raises (::CosLifeCycle::NoFactory,
        ::CosLifeCycle::NotCopyable,
        ::CosLifeCycle::InvalidCriteria,
        ::CosLifeCycle::CannotMeetCriteria);
```

The `copy` operation makes a copy of the node and its roles. The new node and roles should be collocated with the factory finder given by the `there` parameter. The final input parameter, `the_criteria`, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

The result of a `copy` operation is a:

- *Node* object reference for the new node and
- Sequence of roles

Figure 6-18 illustrates the result of a copy. A node, when it is born, is not in any relationships with other objects. That is, the roles in the new node are “disconnected”. It is the compound copy operation’s job to correctly establish new relationships.

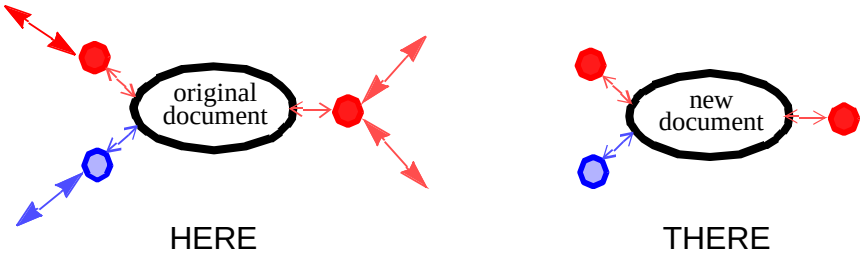

Figure6-18 Copying a node returns the new object and the corresponding roles.

If the node or one of its roles refuses to be copied, the `NotCopyable` exception is raised with the node or role object reference returned as a parameter to the exception.

If an appropriate factory to create a copy cannot be found, the `NoFactory` exception is raised. The exception value indicates the key used to find the factory.

In addition to the `NoFactory` and `NotCopyable` exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the copied node, `NO_RESOURCES` will be raised.

Moving a Node

```
void move_node (in ::CosLifeCycle::FactoryFinder there,
       in ::CosLifeCycle::Criteria the_criteria)
   raises (::CosLifeCycle::NoFactory,
       ::CosLifeCycle::NotMovable,
       ::CosLifeCycle::InvalidCriteria,
       ::CosLifeCycle::CannotMeetCriteria);
```

The `move` operation transfers some or all of the node’s resources from “here” to “there”. The `move` operation migrates a the node and its roles. The migrated node and roles should be collocated with the factory finder given by the `there` parameter. The final parameter, `the_criteria`, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

If the node or one of its roles refuses to be moved, the `NotMovable` exception is raised with the node or role object reference returned as a parameter to the exception.

If an appropriate factory to support migration “over there” cannot be found, the `NoFactory` exception is raised. The exception value indicates the key used to find the factory.

In addition to the `NoFactory` and `NotMovable` exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the migrated node, `NO_RESOURCES` will be raised.

Removing a Node

```
void remove_node ()
    raises (::CosLifeCycle::NotRemovable);
```

The `remove` operation removes the node and its roles.

If the node or one of its roles refuses to be removed, the `NotRemovable` exception is raised with the node or role object reference returned as a parameter to the exception.

Getting the Node's Life Cycle Object

```
::CosLifeCycle::LifeCycleObject get_life_cycle_object()
    raises (NotLifeCycleObject);
```

Some nodes not only participate in the life cycle protocols for graphs of related objects but they also support the client's view of life cycle services. That is, the node also supports the *::CosLifeCycle::LifeCycleObject* interface described in the Life Cycle Service specification. The `get_life_cycle_object` operation returns the *::CosLifeCycle::LifeCycleObject* object reference for the node.

If the node does not support the *::CosLifeCycle::LifeCycleObject* interface, the `NotLifeCycleObject` exception is raised.

The Role Interface

The *Role* interface defines operations to copy and move a role. (The `destroy` operation is defined by the base Relationship Service. As such, there is no need to define a `remove` operation.) The *Role* interface also defines an operation to return the propagation values for the copy, move and remove operations.

The implementation of a *CompoundLifeCycle::Node* operation can call these operations on roles. For example, an implementation of `copy` on a node can call the `copy` operation on the *Role*.

Copying a Role

```
Role copy_role (in ::CosLifeCycle::FactoryFinder there,
        in ::CosLifeCycle::Criteria the_criteria)
    raises (::CosLifeCycle::NoFactory,
        ::CosLifeCycle::NotCopyable,
        ::CosLifeCycle::InvalidCriteria,
        ::CosLifeCycle::CannotMeetCriteria);
```

The `copy` operation makes a copy of the role. The new role should be collocated with the factory finder given by the `there` parameter. The final parameter, `the_criteria`, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

The result of a `copy` operation is an object reference for the new object supporting the *Role* interface.

If the role refuses to be copied, the `NotCopyable` exception is raised with the role object reference returned as a parameter to the exception.

If an appropriate factory to create a copy cannot be found, the `NoFactory` exception is raised. The exception value indicates the key used to find the factory.

In addition to the `NoFactory` and `NotCopyable` exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the copied role, `NO_RESOURCES` will be raised.

Moving a Role

```
void move_role (in ::CosLifeCycle::FactoryFinder there,
        in ::CosLifeCycle::Criteria the_criteria)
    raises (::CosLifeCycle::NoFactory,
        ::CosLifeCycle::NotMovable,
        ::CosLifeCycle::InvalidCriteria,
        ::CosLifeCycle::CannotMeetCriteria);
```

The `move` operation transfers some or all of the role's resources. The `move` operation migrates the role. The migrated role should be collocated with the factory finder given by the `there` parameter. The final parameter, `the_criteria`, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

If the role refuses to be moved, the `NotMovable` exception is raised with the role object reference returned as a parameter to the exception.

If an appropriate factory to support migration cannot be found, the `NoFactory` exception is raised. The exception value indicates the key used to find the factory.

In addition to the `NoFactory` and `NotMovable` exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the migrated role, `NO_RESOURCES` will be raised.

Getting a Propagation Value

```
::CosGraphs::PropagationValue life_cycle_propagation (
        in Operation op,
        in RelationshipHandle rel,
        in ::CosRelationships::RoleName to_role_name,
        out boolean same_for_all);
```

The `life_cycle_propagation` operation returns the propagation value to the role `to_role_name` for the life cycle operation `op` and the relationship `rel`. If the role can guarantee that the propagation value is the same for all relationships in which it participates, `same_for_all` is true.

The Relationship Interface

The *Relationship* interface defines operations to copy and move a relationship. (The `destroy` operation is defined by the Relationship Service. As such, there is no need to define a `remove` operation.) The *Relationship* interface also defines an operation to return the propagation values for the copy, move and remove operations.

Copying the Relationship

```
Relationship copy_relationship (
        in ::CosLifeCycle::FactoryFinder there,
        in ::CosLifeCycle::Criteria the_criteria,
        in ::CosGraphs::NamedRoles new_roles)
    raises (::CosLifeCycle::NoFactory,
        ::CosLifeCycle::NotCopyable,
        ::CosLifeCycle::InvalidCriteria,
        ::CosLifeCycle::CannotMeetCriteria);
```

The `copy` operation creates a new relationship. The new relationship should be collocated with the factory finder given by the `there` parameter. The second parameter, `the_criteria`, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

The values of the newly created relationship's attributes are defined by the implementation of this operation. However, the `named_roles` attribute of the newly created relationship must match new_roles. That is, the newly created relationship relates objects represented by `new_roles` parameter, not the by the original relationship's named roles.

The result of a `copy` operation is an object reference for the new object supporting the *Relationship* interface.

If the relationship refuses to be copied, the `NotCopyable` exception is raised with the relationship object reference returned as a parameter to the exception.

If an appropriate factory to create a copy cannot be found, the `NoFactory` exception is raised. The exception value indicates the key used to find the factory.

In addition to the `NoFactory` and `NotCopyable` exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the copied role, `NO_RESOURCES` will be raised.

Moving the Relationship

```
void move_relationship (
        in ::CosLifeCycle::Criteria the_criteria)
    raises (::CosLifeCycle::NoFactory,
        ::CosLifeCycle::NotMovable,
        ::CosLifeCycle::InvalidCriteria,
        ::CosLifeCycle::CannotMeetCriteria);
```

The `move` operation transfers some or all of the relationship's resources. The `move` operation migrates the relationship. The migrated relationship should be collocated with the factory finder given by the `there` parameter. The final parameter, `the_criteria`, allows unspecified values to be passed. This is explained in the Life Cycle specification in detail.

If the relationship refuses to be moved, the `NotMovable` exception is raised with the relationship object reference returned as a parameter to the exception.

If an appropriate factory to support migration cannot be found, the `NoFactory` exception is raised. The exception value indicates the key used to find the factory.

In addition to the `NoFactory` and `NotMovable` exceptions, implementations may raise standard CORBA exceptions. For example, if resources cannot be acquired for the migrated relationship, `NO_RESOURCES` will be raised.

Getting a Propagation Value

```
::CosGraphs::PropagationValue life_cycle_propagation (
        in Operation op,
        in ::CosRelationships::RoleName from_role_name,
        in ::CosRelationships::RoleName to_role_name,
        out boolean same_for_all);
```

The `life_cycle_propagation` operation returns the relationship's propagation value from the role `from_role` to the role `to_role_name` for the life cycle operation `op`. If the role named by `from_role_name` can guarantee that the propagation value is the same for all relationships in which it participates, `same_for_all` is true.

The PropagationCriteriaFactory Interface

The *CosGraphs* module in the Relationship Service defines a general service for traversing a graph of related objects. The service accepts a "call-back" object supporting the *::CosGraphs::TraversalCriteria* interface. Given a node, this object defines which edges to emit and which nodes to visit next.

The *PropgationCriteriaFactory* creates a *TraversalCriteria* object that determines which edges to emit and which nodes to visit based on propagation values for the compound life cycle operations.

Create a Traversal Criteria Based on Life Cycle Propagation Values

```
::CosGraphs::TraversalCriteria create(in Operation op);
```

The `create` operation returns a TraversalCriteria object for an operation op that determines which edges to emit and which nodes to visit based on propagation values for op. For a more detailed discussion see section A.4.1 of this appendix and section 9.4.2 of the Relationship specification.

A.4.4 Specific Life Cycle Relationships

The Relationship service defines two important relationships, *containment* and *reference*. Containment is a one-to-many relationship. A container can contain many containees; a containee is contained by one container. Reference, on the other hand, is a many-to-many relationship. An object can reference many objects; an object can be referenced by many objects.

Containment is represented by a relationship with two roles: the *ContainsRole*, and the *ContainedInRole*. Similarly, reference is represented by a relationship with two roles: *ReferencesRole and ReferencedByRole*.

The compound life cycle specification adds life cycle semantics to these specific relationships. That is, it defines propagation values for containment and reference.

A.4.5 The CosLifeCycleContainment Module

The *CosLifeCycleContainment* module defines three interfaces

- the *Relationship* interface
- the *ContainsRole* interface and
- the *ContainedInRole* interface.

```
#include <Containment.idl>
#include <CompoundLifeCycle.idl>

module CosLifeCycleContainment {

    interface Relationship :
        ::CosCompoundLifeCycle::Relationship,
        ::CosContainment::Relationship {};

    interface ContainsRole :
        ::CosCompoundLifeCycle::Role,
        ::CosContainment::ContainsRole {};

    interface ContainedInRole :
        ::CosCompoundLifeCycle::Role,
        ::CosContainment::ContainedInRole {};
};
```

Figure6-19 The CosLifeCycleContainment module

The *CosLifeCycleContainment* module does not define new operations. It merely "mixes in" interfaces from the *CosCompoundLifeCycle* and *CosContainment* modules. Although it does not add any new operations, it refines the semantics of these attributes and operations:

RelationshipFactory attribute	**value**
`relationship_type`	`CosLifeCycleContainment::Relationship`
`degree`	`2`
`named_role_types`	`"ContainsRole",CosLifeCycleContainment::ContainsRole;` `"ContainedInRole",CosLifeCycleContainment::ContainedInRole`

The CosRelationships::RelationshipFactory::`create` operation will raise `DegreeError` if the number of roles passed as arguments is not 2. It will raise `RoleTypeError` if the roles are not *CosLifeCycleContainment*::*ContainsRole* and *CosLifeCycleContainment*::*ContainedInRole*. It will raise `MaxCardinalityExceeded` if the *CosLifeCycleContainment*::*ContainedInRole* is already participating in a relationship.

RoleFactory attribute for ContainsRole	**value**
`role_type`	`CosLifeCycleContainment::ContainsRole`
`maximum_cardinality`	`unbounded`
`minimum_cardinality`	`0`
`related_object_types`	`CosCompoundLifeCycle::Node`

The *CosRelationships::RoleFactory::*`create_role` operation will raise the `RelatedObjectTypeError` if the related object passed as a parameter does not support the *CosCompoundLifeCycle::Node* interface. The *CosRelationships::RoleFactory::*`link` operation will raise `RelationshipTypeError` if the `rel` parameter does not conform to the *CosLifeCycleContainment::Relationship* interface.

RoleFactory attribute for ContainedInRole	**value**
`role_type`	`CosLifeCycleContainment::ContainedInRole`
`maximum_cardinality`	`1`
`minimum_cardinality`	`1`
`related_object_types`	`CosCompoundLifeCycle::Node`

The *CosRelationships::RoleFactory::*`create_role` operation will raise the `RelatedObjectTypeError` if the related object passed as a parameter does not support the *CosCompoundLIfeCycle::Node* interface. The *CosRelationships::RoleFactory::*`link` operation will raise `RelationshipTypeError` if the `rel` parameter does not conform to the *CosLifeCycleContainment::Relationship* interface. The *CosRelationships::RoleFactory::*`link` operation will raise `MaxCardinalityExceeded` if it is already participating in a containment relationship.

The *CosLifeCycleContainment*::*ContainsRole*::`life_cycle_propagation` operation returns the following:

operation	ContainsRole to ContainedInRole
copy	deep
move	deep
remove	deep

The *CosLifeCycleContainment*::*ContainedInRole*::`life_cycle_propagation` operation returns the following::

operation	ContainedInRole to ContainsRole
copy	shallow
move	shallow
remove	shallow

A.4.6 The CosLifeCycleReference Module

The *CosLifeCycleReference* module defines three interfaces

- the *Relationship* interface,
- the *ReferencesRole* interface and
- the *ReferencedByRole* interface.

```
#include <Reference.idl>
#include <CompoundLifeCycle.idl>

module CosLifeCycleReference {

    interface Relationship :
        ::CosCompoundLifeCycle::Relationship,
        ::CosReference::Relationship {};

    interface ReferencesRole :
        ::CosCompoundLifeCycle::Role,
        ::CosReference::ReferencesRole {};

    interface ReferencedByRole :
        ::CosCompoundLifeCycle::Role,
        ::CosReference::ReferencedByRole {};
};
```

Figure6-20 The CosLifeCycleReference module

The *CosLifeCycleReference* module does not define new operations. It merely “mixes in” interfaces from the *CosCompoundLifeCycle* and *CosReference* modules. Although it does not add any new operations, it refines the semantics of these attributes and operations:

RelationshipFactory attribute	**value**
`relationship_type`	`CosLifeCycleReference::Relationship`
`degree`	`2`
`named_role_types`	`“ReferencesRole”,CosLifeCycleReference::ReferencesRole;` `“ReferencedByRole”,CosLifeCycleReference::ReferencedByRole`

The CosRelationships::RelationshipFactory::`create` operation will raise `DegreeError` if the number of roles passed as arguments is not 2. It will raise `RoleTypeError` if the roles are not *CosReference::ReferencesRole* and *CosReference*::*ReferencedByRole*.

RoleFactory attribute for ReferencesRole	**value**
`role_type`	`CosLifeCycleReference::ReferencesRole`
`maximum_cardinality`	`unbounded`
`minimum_cardinality`	`0`
`related_object_types`	`CosCompoundLifeCycle::Node`

The *CosRelationships::RoleFactory::*`create_role` operation will raise the `RelatedObjectTypeError` if the related object passed as a parameter does not support the *CosCompoundLifeCycle::Node* interface. The *CosRelationships::RoleFactory::*`link` operation will raise `RelationshipTypeError` if the `rel` parameter does not conform to the *CosLifeCycleReference::Relationship* interface.

RoleFactory attribute for ReferencedByRole	**value**
`role_type`	`CosLifeCycleReference::ReferencedByRole`
`maximum_cardinality`	`unbounded`
`minimum_cardinality`	`0`
`related_object_types`	`CosCompoundLifeCycle::Node`

The *CosRelationships::RoleFactory::*`create_role` operation will raise the `RelatedObjectTypeError` if the related object passed as a parameter does not support the *CosCompoundLifeCycle::Node* interface. The

CosRelationships::RoleFactory::`link` operation will raise `RelationshipTypeError` if the `rel` parameter does not conform to the *CosLifeCycleRelationship::Relationship* interface.

The *CosLifeCycleReference*`::`*ReferencesRole*`::life_cycle_propagation` operation returns the following:

operation	**ReferencesRole to ReferencedByRole**
copy	shallow
move	shallow
remove	shallow

The *CosLifeCycleReference*`::`*ReferencedByRole*`::life_cycle_propagation` operation returns the following::

operation	**ReferencedByRole to ReferencesRole**
copy	none
move	shallow
remove	shallow

The *CosRelationships::RoleFactory::*`create_role` operation will raise the `RelatedObjectTypeError` if the related object passed as a parameter does not support the *CosCompoundLifeCycle::Node* interface.
The CosRelationships::RelationshipFactory::`create` operation will raise `DegreeError` if the number of roles passed as arguments is not 2. It will raise `RoleTypeError` if the roles are not *CosLifeCycleReference*::*ReferencesRole* and *CosLifeCycleReference*::*ReferencedByRole*.

A.5 References

1. James Rumbaugh, “Controlling Propagation of Operations using Attributes on Relations.” *OOPSLA 1988 Proceedings*, pg. 285-296

2. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, “Object-oriented Modeling and Design.” Prentice Hall, 1991.

Appendix B Filters

Note – Appendix B is not part of the Life Cycle Services specification. It sketches a mechanism for expressing filters. This appendix is included to provided an example of how a filter might be provided.

A factory represents a scope of resource allocation, which is the set of resources available to the factory. Whenever it receives a creation request, a factory will allocate resources according to any policies which are in operation.

Clearly, by choosing a particular factory upon which to issue a create request, a client is exerting some control over the allocation of resources. Therefore, a client can limit the scope of resource allocation, by issuing the request on a different factory which represents a smaller set of resources.

However, there are two problems with this. Firstly, the granularity of resources may be much smaller than the granularity represented by the factories in a system. For example, there are unlikely to be factories which represent individual disk segments.

Secondly, the client may wish to rule out the use of particular resources within a scope, but avoid having a general reduction in scope. For example, the client might not be concerned with which machine within a LAN an object is created on, providing it is not on machine X.

Both of these needs can be addressed by providing a filter. In the first case, the filter is relatively simple; it will simply limit the scope of resource allocation. In the second case, the filter will need to be more sophisticated.

This appendix describes one way of providing filters using *properties* and *constraint expressions*. These concepts appear in the development of Trading in the ISO/IEC/CCITT Open Distributed Processing standards. Service providers register their service with the Trader and use properties to describe the service offer. Potential clients may then use a constraint expressions to describe the requirements which service offers must satisfy.

Similarly, the life cycle service may define a number of properties to represent the different kinds of resources available within in a system and clients may use constraint expressions to place the restrictions upon the use of those resources.

Note – The Object Services Architecture identifies an Object Properties Service which enables an object to have a set of arbitrary named values associated with it. These are very similar to the concept of properties as used in Trading and in this appendix.

B.1 Resources as Properties

Resource properties are application and generic factory implementation dependent and it is beyond the scope of this specification to identify standard properties which all generic factory implementations will recognize. The properties described in this appendix are given as examples only. Table 6-8 gives some examples of properties that might be supported by a generic factory.

Table 6-8 Examples of properties supported by a generic factory

Property Name	Meaning
Host	Host name of the machine
Architecture	Machine architecture, e.g. “intel”, “sparc”
OSArchitecture	Operating system architecture e.g. “solaris”, “hpux”

B.2 Constraint Expressions

Constraints are expressed in a Constraint Language which provides a set of operators which allow arbitrarily complex expressions involving properties and potential values to be specified. A property lists *satisfies* a constraint if the constraint expression is true when evaluated with respect to the property list.

Constraint expressions are very flexible. For example, if a client has an object executing on a machine called ‘Host1’ and wishes to create another object which is *not* on the same machine, the client can specify the constraint “Host != ‘Host1’”.

The constraint expression described here works with properties for which the value can be a string, a number, or a set of values.

The constraint language consists of:

- comparative functions: `==`, `!=`, `>`, `>=`, `<`, `<=`, `in`
- constructors: `and`, `or`, `not`
- property names
- numeric and string constants
- mathematical operators: `+`, `-`, `*`, `/`
- grouping operators: `(`, `)`, `[`, `]`

The following precedence relations hold in the absence of parentheses, in the order of lowest to highest:

- `+` and `-`
- `*` and `/`
- `or`
- `and`
- `not`

The comparative operator `in` checks for the inclusion of a particular string constant in the list which is the value of a property.

B.3 BNF for Constraint Expressions

```
<ConstraintExpr>    :=      [ <Expr> ]

<Expr>              :=      <Expr> ”or” <Expr>
                    |       <Expr> ”and” <Expr>
                    |       ”not” <Expr>
                    |       ”(” <Expr> ”)”
                    |       <SetExpr> <SetOp> <SetExpr>
                    |       <StrExpr> <StrOp> <StrExpr>
                    |       <NumExpr> <NumOp> <NumExpr>
                    |       <NumExpr> ”in” <SetExpr>
                    |       <StrExpr> ”in” <SetExpr>

<NumOp>             :=      ”==” | ”!=” | ”<” | ”<=” | ”>” | ”>=”

<StrOp>             :=      ”==” | ”!=”

<SetOp>             :=      ”==” | ”!=”


<NumExpr>           :=      <NumTerm>
                    |       <NumExpr> ”+” <NumTerm>
                    |       <NumExpr> ”-” <NumTerm>

<NumTerm>           :=      <NumFactor>
                    |       <NumTerm> ”*” <NumFactor>
                    |       <NumTerm> ”/” <NumFactor>

<NumFactor>         :=      <Identifier>
                    |       <Number>
                    |       ”(” <NumExpr> ”)”
                    |       ”-” <NumFactor>

<StrExpr>           :=      <StrTerm>
                    |       <StrExpr> ”+” <StrTerm>

<StrTerm>           :=      <Identifier>
                    |       <String>
                    |       ”(” <StrExpr> ”)”

<SetExpr>           :=      <SetTerm>
                    |       <SetExpr> ”+” <SetTerm>

<SetTerm>           :=      <Identifier>
                    |       <Set>
                    |       ”(” <SetExpr> ”)”

<Identifier>        :=      <Word>
```

```
<Number>          :=    <Integer>
                  |     <Float>

<Integer>         :=    { <Digit> }+

<Float>           :=    <Mantissa> [ <Sign> ] [ <Exponent> ]

<Mantissa>        :=    <Integer> [ ”.” [ <Integer> ] ]
                  |     ”.” <Integer>

<Sign>            :=    ”-”
                  |     ”+”

<Exponent>        :=    ”e” <Integer>
                  |     ”E” <Integer>

<Word>            :=    <Letter> { <AlphaNum> }*

<AlphaNum>        :=    <Letter>
                  |     <Digit>
                  |     ”_”

<String>          :=    ”’” { <Char> }* ”’”

<Char>            :=    <Letter>
                  |     <Digit>
                  |     <Other>

<Set>             :=    ”{” <Elements> ”}”

<Elements>        :=    [ <Element> { <Sp>+ <Element> }* ]

<Element>         :=    <Number>
                  |     <Word>
                  |     <String>

<Letter>          :=    a | b | c | d | e | f | g | h | i | j | k
                  |     l | m | n | o | p | q | r | s | t | u | v
                  |     w | x | y | z | A | B | C | D | E | F | G
                  |     H | I | J | K | L | M | N | O | P | Q | R
                  |     S | T | U | V | W | X | Y | Z

<Digit>           :=    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Other>           :=    <Sp> | ~ | ! | @ | # | $ | % | ^ | & | * | (
                  |     ) | - | _ | = | + | [ | { | ] | } | ; | :
                  |     “ | \ | | | , | < | . | > | / | ?

<Sp>              :=    ” ”
```

Appendix C Administration

Note – Appendix C is not part of the Life Cycle Services specification. This description is included as a suggested way of administering generic factories.

The specification for the life cycle service includes the *GenericFactory* interface. There will be at least two styles of object which support that interface:

- implementation specific factories that actually assemble the resources for a new object, and
- generic factories which pass requests on to either implementation specific factories or other generic factories.

By configuring generic factories and implementation specific factories into a graph, a creation service can be built which administers the allocation of a large number of resources and can use them to create a wide variety of objects.

To ensure that the creation service is scalable, it is essential that the principle of *federation* is adopted – each component retains its autonomy rather than becoming subordinate to another.

Whenever the creation service receives a creation request, the request will need to traverse the graph until it reaches an implementation specific factory which can satisfy the request. As the request traverses the graph, each non-terminal node in the graph (i.e. the generic factories) will decide which link the request will traverse next. Decisions will be based upon information about each available link, any *policies* in force at that node and, of course, the actual request.

Clearly, the configuration and policies of such a creation service will need to be administered. However, the specification does not include the specification of an administration interface. This is because the principle of federation is not only important to the life cycle service. It will be essential to a number of other services, notably trading, and the OMG plans to address the issue of federation for *all* object services, rather than making a premature specification addressing the needs of just one service.

The remainder of this appendix describes the principle of federation in more detail, outlines the use of policies and preferences to support federation, and then concludes with a suggestion for how an administration interface might look.

C.1 Federation

Federation is essential in large-scale distributed systems where the existence of centralized ownership and universal control cannot be assumed. In these systems the only way to achieve cooperation between autonomous systems without creating a hierarchical structure is to use federation. Federation is also beneficial to smaller systems which can exploit the high degree of flexibility which federation provides.

Federation differs from the more conventional approach of adopting a strictly hierarchical organization in a number of ways. Firstly, components can provide their service to any number of others, not just the single component which is its “parent” in the hierarchy. Secondly, components can establish peer-to-peer relationships, eliminating the need for a single component at the top of the hierarchy. Finally, this approach avoids the necessity of maintaining a global namespace. Instead, all names are relative to the context in which they are used.

Federation enables previously distinct systems to be unified without requiring global changes to their naming structures and system management hierarchies. The administration functions must ensure the systems are configured appropriately, e.g. avoiding circular references in those graphs which must be kept acyclic.

C.1.1 Federation in Object Services

In addition to the use of federation in configuring generic factories, federation is also applicable to a number of other services.

Trading is a notable example. A global offer space is neither practical nor desirable. Consequently, there will be multiple traders, each representing a different portion of the offer space. Offers held by one trader can be made available to the clients of another trader through federation.

The naming service specification also demonstrates attributes of federation. Naming contexts can be bound to other naming contexts and requests for name resolution can be passed across the links. However, it is entirely the concern of the naming context how it resolves the name within its domain, i.e. it is autonomous.

C.1.2 Federation Issues

There are a number of issues which need to be addressed for federation to be used in a cohesive fashion across all object services.

Visibility of the Federation Graph

The naming service makes the configuration of naming contexts into a graph very visible to the clients. This is essential, because the naming service must provide clients with a structured namespace.

On the other hand, it is not clear that a client should ever be able to see the internal structure of a life cycle creation service built with generic and implementation specific factories.

The trading service falls in between the two extremes. It may be useful for a client to be able to navigate the structure of a trading service graph in order to have more control over the visibility of offers. However, this may make clients too dependent upon the organization of the trading service and limit the flexibility of the system administrator in reorganizing the trading service to provide the most effective service.

Service Interface vs. Administration Interface

In general, it is desirable to federate using the service interface for the links and reserve the administration interface for the administrators. This approach ensures that autonomy is retained. However, this precludes the use of compound names in the administration functions because the administration functions cannot traverse the graph; only simple names can be used in administration only functions.

However, this is inappropriate for services where graph manipulation is an essential part of the service. For example, the naming service specification does not distinguish between administration functions for manipulating the graph and service functions. This is clearly correct; the clients need to be able to manipulate the graph by creating, binding and destroying contexts.

Multiple Service Interfaces

A node in a federation graph may be a conspiracy and offer multiple service interfaces, perhaps one for each point it is bound into the graph. However, for services where the administration is kept distinct from the service, it is likely that the conspiracy will support only one administration interface.

In these situations, it becomes necessary for an administrator to be able to match service interfaces to conspiracies, i.e. to match one or more service interfaces to an administrative interface. The example in Section C.3 provides a solution to this which, in theory, will scale, but there may be better ways of doing this.

Cycles and Peer-to-Peer Relationships

The introduction of cycles into a federation graph is a contentious issue. Since peer-to-peer relationships are a degenerate form of cycle, any service which supports peer-to-peer relationships must be capable of handling cycles. The major impact of this is to provide loop detection on operations which would otherwise go out of control. Both trading and naming services are examples of this kind of service.

However, some services may not be able to handle cycles effectively and will wish to proscribe them. This probably covers peer-to-peer relationships, although that might be an acceptable special case. An example of this might be the life cycle creation service, where information about the current usage of the available resources must percolate up the graph in order to make informed decisions, but the introduction of cycles would make this information unclear or even meaningless.

C.2 Policies

It is frequently necessary to configure the way in which operations are performed in order to tune the performance, e.g how long a search operation may take, how many matches can be returned, or how much memory to use for a cache.

The same problems exist in distributed systems except that such configuration parameters must be explicitly passed around. Where different administrative domains are connected, such configuration parameters cannot be enforced by one domain on the other. Similarly, users may want to control the configuration but must be prevented from hogging resources, e.g memory, disk space, etc. Some configuration elements must be enforced, e.g disk quotas, some elements may specify defaults which can be changed and some elements may be requests which may or may not clash with hard limits e.g max memory per process.

Policies are used as a generic solution to this problem – wherever some kind of choice needs to be made, policies may be used to guide the decision making process.

Table 6-9 provides some examples of policies. which a federated service might support.

Table 6-9 Example policies

Policy Name	Meaning
search_algorithm	determines whether the federation graph should be traversed in a depth first or breadth first fashion.
cross_ boundaries	determines whether administrative boundaries should be crossed.
maximum_distance	how far to traverse a graph before failing a request.

When invoking operations, clients can specify preferences for particular policies. Providing the service has no value set for that policy, the preference will be simply added to the policy list for the duration of the request. However, if a service policy is already specified then the preference will either be ignored or, for policies such as “maximum_distance”, the more constraining value will be adopted.

As a request traverses a graph, each node will pass its current policy set as preferences. In this way, the autonomy of individual administrative domains is preserved.

When an object doesn’t implement all choices of a policy, it should not allow its policy to be modified to an unsupported value. This means that implementation limitations are handled as Administrative hard limits which provides the correct semantics.

Where no policy is specified by either administrator or client, the implementation determines its own behavior. However, this decision would not be propagated through the graph (as a preference), leaving it to each node in the graph to make its own decision.

C.3 An Example LifeCycleService Module

Administrators access the administration functions via the *LifeCycleService* module, which defines the *LifeCycleServiceAdmin* interface. This example is intended to work with the *GenericFactory* interface in the specification. As a result, the administration functions cannot make use of compound names.

```
#include "LifeCycle.idl"

module LifeCycleService {

    typedef sequence <Lifecycle::NameValuePair> PolicyList;
    typedef sequence <Lifecycle::Key> Keys;
    typedef sequence <Lifecycle::NameValuePair> PropertyList;
    typedef sequence <Naming::NameComponent> NameComponents;

    interface LifeCycleServiceAdmin {

        attribute PolicyList policies;

        void bind_generic_factory(
                in Lifecycle::GenericFactory gf,
                in Naming::NameComponent name,
                in Keys key_set,
                in PropertyList other_properties)
            raises (Naming::AlreadBound, Naming::InvalidName);

        void unbind_generic_factory(
                in Naming::NameComponent name)
            raises (Naming::NotFound, Naming::InvalidName);

        Lifecycle::GenericFactory resolve_generic_factory(
                in Naming::NameComponent name)
            raises (Naming::NotFound, Naming::InvalidName);

        NameComponents list_generic_factories();

        boolean match_service (in Lifecycle::GenericFactory f);

        string get_hint();

        void get_link_properties(
                in Naming::NameComponent name,
                out Keys key_set,
                out PropertyList other_properties)
            raises (Naming::NotFound, Naming::InvalidName);
    };

};
```

Figure 6-20 The LifeCycleService Module

C.3.1 The LifeCycleServiceAdmin Interface

The *LifeCycleServiceAdmin* interface provides the basic administration operations required to enable the lifecycle service to be administered by a set of tools or an administration service. The operations enable configuration of factories supporting the *GenericFactory* interface into a graph and setting of policies for those factories.

bind_generic_factory

```
void bind_generic_factory(
        in Lifecycle::GenericFactory gf,
        in Naming::NameComponent name,
        in Keys key_set,
        in PropertyList other_properties)
    raises (Naming::AlreadBound, Naming::InvalidName);
```

This operation binds a factory supporting the *GenericFactory* interface into a graph. The `name` must be unique within the context of the target of the operation. From then on, that factory can be identified by that name.

In order to make a good decision about which link to choose for a request, the node needs to be provided with additional information about those factories. This information may be fairly dynamic, e.g. the current usage of the resources available through the link, or more static, e.g. the Keys for which the link can provide support.

The `key_set` parameter is a list of the keys for which the factory can provide support. In the case of an implementation specific factory, this list will often only have one member.

The `other_properties` parameter can be used to provide other static properties associated with the factory. For example, an “Architectures” property would indicate the type(s) of machine which the factory could create objects on.

Changes to the static information as well as more dynamic information can be monitored through the Events service. Each factory would generate events whenever the information changed significantly (e.g. a new *GenericFactory* interface with new keys is bound to the factory, or there is a change in the usage of resources available to the factory) and these can then be passed to those factories which need to know.

unbind_generic_factory

```
void unbind_generic_factory(
        in Naming::NameComponent name)
    raises (Naming::NotFound, Naming::InvalidName);
```

This operation unbinds the generic factory identified by the name.

resolve_generic_factory

```
Lifecycle::GenericFactory resolve_generic_factory(
        in Naming::NameComponent name)
    raises (Naming::NotFound, Naming::InvalidName);
```

This operation takes the name supplied and returns the reference to the *GenericFactory* object.

list_generic_factories

```
NameComponents list_generic_factories();
```

This operation returns a list of the names of all the bound factories.

match_service

```
boolean match_service (in Lifecycle::GenericFactory f);
```

This operation returns `true` if the generic factory interface is supported by the target.

get_hint

```
string get_hint();
```

This operation returns a hint associated with the target, see *Building a Map of a Graph* below.

get_link_properties

```
void get_link_properties(
        in Naming::NameComponent name,
        out Keys key_set,
        out PropertyList other_properties)
    raises (Naming::NotFound, Naming::InvalidName);
```

This operation returns the `key_set` and `other_properties` associated with the `name`.

Building a Map of a Graph

Administration tools may wish to build a map of a federation graph from scratch and some of the operations above are provided for that purpose.

First of all, the tool must obtain the set of administration interfaces for all the factories to be administered. These might be obtained from a number of sources, e.g. a well-known trading context.

For each interface, the `list_generic_factories` operation obtains a list of all the links for each node. Using `resolve_generic_factory`, a service interface can be obtained for each link. These can then be matched to an administration interface using `match_service`.

Clearly, this does not scale well if there are many nodes involved because of the average number of invocations of `match_service` required. This problem can be solved if one of the `other_properties` associated with each service interface is a *hint* and a hint is available for each administration interface. If the hints are the same, there may be a match and `match_service` is called to check. If the hints could be guaranteed to be unambiguous, the invocation could be avoided altogether, but this requires a global namespace for the hints. The best that can reasonably be achieved is to reduce the chance of a clash to a minimum.

The `get_hint` and `get_link_properties` can be used for this purpose.

Appendix D Support for PCTE Objects[5]

Note – Appendix D is not part of the Life Cycle Services specification. This appendix defines a set of criteria[6] suitable for supporting PCTE objects.

It is intended that objects in a PCTE repository be among those objects that can be managed though this lifecycle interface. It is reasonable to expect that applications written for PCTE will use the PCTE APIs to manage the life-cycle of PCTE objects. It is also reasonable to expect that clients not specifically written for relationship-oriented objects will not be able to manipulate the life-cycles of PCTE objects. However, between these two, one can envision clients which desire to be flexible, working on objects which may or may not be stored in the PCTE repository. One can also envision object factories, constructed to make use of PCTE which provide services to clients that are not PCTE applications because they do not have the appropriate working schemas, etc.

Support for these clients employs a series of conventional interpretations of the lifecycle operations. This appendix provides one such set of conventions to demonstrate the feasibility of the use of these interfaces in a context supporting PCTE.

Object references appear in constraint expressions in the form of character strings. Any implementation of PCTE as a CORBA Object Adapter has to establish a relationship between these and the corresponding CORBA types, and be able to convert between them.

D.1 Overview

A PCTE repository can be viewed as a generic factory. Using whatever naming or trading services are appropriate, a client wishing to use the PCTE factory obtains an object reference to it. To support the simple applications intending to operate within the context of a single PCTE repository, the PCTE factory supports the operations defined by both the *GenericFactory* and *FactoryFinder* interfaces. The client can then invoke the PCTE factory's `create_object` operation, or pass the factory as the "factory finder" when invoking the move or copy operations to move or copy within the same PCTE repository. These clients include the servers implementing the `move` and `copy` operations for various PCTE objects as well.

5. PCTE details used here are from the PCTE Abstract Specification, Standard ECMA-149 available from the European Computer Manufacturers Association.

6. As defined in section 6.2.4 of the life cycle specification.

Lifecycle creation, copy, and move operations are influenced by a sequence of criteria. Criteria are specified as a sequence of name/value pairs. Certain criteria are of interest to the PCTE factories:

"logical location"

The logical location is used to express the logical connection information that must be specified when creating or copying a PCTE object. Logical location is a sequence of name/value pairs expressing a connection for the object. The PCTE factory supports and requires two:

ORIGIN	A string representation of the reference to the object to which the newly created object is to be connected.
ORIGINLINK	The name of the origin object's link which is to hold the link from the origin object to the newly created object.

"filter"

The filter is used to express the fact that an object being created, copied, or moved should reside on the same volume as some other, nearby, object. A filter is an expression as described in B.3. For PCTE, the term "NEAR=" followed by an object reference to the designated nearby object indicates that the new object is to be located at least as near as the same volume to the specified object. "authorization" Although omitted from table 1-4 because no proposal on authorization has yet been accepted by OMG, this lifecycle criterion is required to create PCTE objects.

D.2 Object Creation

The `LifeCycle::GenericFactory::create_object` operation in this specification is borne by factory objects. It has two parameters:

1. a key used to identify the desired object to be created and
2. a set of criteria expressed in an NVP-list.

The corresponding PCTE operation is called OBJECT_CREATE. The parameters to OBJECT_CREATE are obtained from the `LifeCycle::GenericFactory::create_object` parameters.

The PCTE operation OBJECT_CREATE has six parameters:

1. the type of object to be created This is the "key" from LifeCycle `create_object`.
2. the origin object of the relation anchoring the new object This is the object identified as the named "ORIGIN" of the logical location criterion.
3. the name of the link from that origin object to the new object This is the string identified as the named "ORIGINLINK" of the logical location criterion.
4. an optional key for that link This is the string identified as the named "LINKKEY" of the initialization criteria.

5. an object near whose location the object is to be created This is the string value of a required filter expression value by the qualifier “NEAR”.

6. an access mask This is the string identified as the named “ACCESS” of the authorization criteria This string is a simple mapping of the granted and denied access rights.

Exceptions raised by PCTE are mapped to suitable LifeCycle exceptions.

D.3 Object Deletion

The `LifeCycle::LifeCycleObject::remove` operation in this specification is borne by all life-cycle objects. It has no parameters.

The corresponding PCTE operation is called OBJECT_DELETE. The parameters to OBJECT_DELETE are obtained from the object to be deleted using information about that object defined in PCTE’s schema information about the object.

The PCTE operation OBJECT_DELETE has two parameters:

1. the origin object of a relation anchoring the object to be deleted and

2. the name of the link from that origin object to the object to be deleted.

To both ensure that the controlling object is actually deleted and maintain the PCTE referential integrity constraints the following steps are performed for each reversible link emanating from the controlling object:

1. Determine the object, o, that the link refers to.

2. Determine the name, r&prime., of the reverse link back from o.

3. Perform PCTE OBJECT_DELETE(o, r&prime.)

The objective is accomplished when all outgoing, reversible links have been dealt with thus, or before that if one of the OBJECT_DELETE calls fails because the object has already been deleted.

Exceptions raised by PCTE are mapped to suitable LifeCycle exceptions.

D.4 Object Copying

The `LifeCycle::LifeCycleObject::copy` operation in this specification is borne by all life-cycle objects. It has two parameters:

1. a factory-finder to assist in locating a factory that provides resources for the copied object

2. a set of criteria expressed in an NVP-list

The corresponding PCTE operation is called OBJECT_COPY. Some of the parameters to OBJECT_COPY can be obtained directly from the LifeCycle copy parameters. Other required information is obtained from the constraint expression parameter of the LifeCycle copy.

The PCTE operation OBJECT_COPY has six parameters:

1. the object to be copied This is the bearer object of LifeCycle copy operation.
2. the origin object of the relation anchoring the new object This is the object identified as the named “ORIGIN” of the logical location criterion.
3. the name of the link from that origin object to the new object This is the string identified as the named “ORIGINLINK” of the logical location criterion.
4. an optional key for that link This is the string identified as the named “LINKKEY” of the initialization criteria.
5. an object near whose location the object is to be created This is the string value of a required filter expression value by the qualifier “NEAR”.
6. an access mask This is the string identified as the named “ACCESS” of the authorization criteria This string is a simple mapping of the granted and denied access rights.

The semantics of the copy operation corresponds to the PCTE OBJECT_COPY semantics. They are based upon details of the object types involved, including which attributes, links and destination objects are “duplicable”.

Exceptions raised by PCTE are mapped to suitable CORBA standard exceptions.

D.5 Object Moving

The `LifeCycle::LifeCycleObject::move` operation in this specification is borne by all life-cycle objects. It has two parameters:

1. a factory-finder to assist in locating a factory that provide resources for the moved object
2. a set of criteria expressed in an NVP-list

The corresponding PCTE operation is called OBJECT_MOVE. The parameters to OBJECT_MOVE can be obtained directly from the LifeCycle copy parameters or from defaults.

The PCTE operation OBJECT_MOVE has three parameters:

1. the object to be copied This is the bearer object of LifeCycle move operation.
2. an object near whose location the object is to be created This is the string value of a required filter expression value by the qualifier “NEAR”.
3. scope - whether to move the object itself or the object and all its components

This will be defaulted to ATOMIC.